

Title: *Structure_threader*: an Improved Method for Automation Parallelization of Programs STRUCTURE, FASTSTRUCTURE and *MaverickK* on Multi Core CPU systems.

Authors: Pina-Martins, Francisco^{ab}; Silva, Diogo N^{ad}; Fino, Joana^{ac}; Paulo, Octávio S.^a

Addresses:

^aComputational Biology and Population Genomics Group, Centre for Ecology, Evolution and Environmental Changes, Departamento de Biologia Animal, Faculdade de Ciências, Universidade de Lisboa, Campo Grande, 1749-016 Lisboa, Portugal

^bDepartamento de Biologia e CESAM, Univ. de Aveiro, Portugal

^cDepartment of Human Genetics, National Institute of Health Dr Ricardo Jorge, Lisbon, Portugal

^dInstituto Superior de Agronomia, Universidade de Lisboa, CIFC - Centro de Investigação das Ferrugens do Cafeeiro, Oeiras, PT.

Keywords: Bioinformatics/Phyloinformatics, Genomics/Proteomics, Molecular Evolution, Population Genetics – Empirical, Clustering, Parallel computing

Corresponding author: Francisco Pina-Martins; Faculdade de Ciências da Universidade de Lisboa, Campo Grande, Ed. C2, Sala 2.3.22, 1749-016 Lisboa, Portugal; f.pinamartins@gmail.com

Running title: *Structure_threader*

20 1. Abstract

21 *Structure_threader* is a program to parallelize multiple runs of genetic clustering
22 software that does not make use of multi-threading technology (STRUCTURE,
23 FASTSTRUCTURE and *Maverick*) on multi-core computers. Our approach was
24 benchmarked across multiple systems and displayed great speed improvements
25 relative to the single threaded implementation, scaling very close to linearly with the
26 number of physical cores used.

27 *Structure_threader* was compared to previous software written for the same task -
28 *ParallelStructure* and *StrAuto*, and was proven to be the faster (up to 25% faster)
29 wrapper under all tested scenarios.

30 Furthermore, *Structure_threader* can perform several automatic and convenient
31 operations, assisting the user in assessing the most biologically likely value of 'K' via
32 implementations such as the "Evanno", or "Thermodynamic Integration" tests and
33 automatically draw the "meanQ" plots (static or interactive) for each value of K (or
34 even combined plots).

35 *Structure_threader* is written in python 3 and licensed under the GPLv3. It can be
36 downloaded free of charge at https://github.com/StuntsPT/Structure_threader.

37 2. Introduction

38 Clustering analyses are widely used in population genetics and, nowadays,
39 population genomics. This technique of using multilocus genotype data to infer
40 population clusters, is frequently performed based on multiple MCMC re-sampling.

41 One of the most popular tools for performing this type of analyses is STRUCTURE
42 (Pritchard, Stephens, & Donnelly, 2000). Despite producing robust results, this
43 approach demands long run times, even in modern machines. This problem is
44 aggravated as the type of analysed datasets, which gradually grow from relatively
45 small, such as microsatellite loci (De Barro, 2005; Muchadeyi et al., 2007), to high
46 throughput sequencing (Lamaze, Sauvage, Marie, Garant, & Bernatchez, 2012;
47 Renault, Grassa, Moyers, Kane, & Rieseberg, 2012), consequently increasing run
48 times by orders of magnitude.

49 The process can be sped up by either running multiple instances of the used
50 software, which is an inefficient and error prone method requiring constant attention
51 and intervention from the user. There are faster software alternatives to STRUCTURE,
52 which can also be used to speed up the analysis process.

53 One such option is analysing the data in the program FASTSTRUCTURE (Raj,
54 Stephens, & Pritchard, 2014), which decreases run times by up to two orders of
55 magnitude. However FASTSTRUCTURE does not support the popular “no admixture”
56 model present in STRUCTURE, and is not capable of handling haploid data (and
57 several other less widely used features), which limits its application to a wide range
58 of data.

59 Another option in to analyse the data in the software *MaverickK* (Verity & Nichols,
60 2016), which is also considerably faster than STRUCTURE, but not as fast as
61 FASTSTRUCTURE by an order of magnitude. It does, however support most of the
62 same features as STRUCTURE and uses a built-in, improved method for helping
63 determine the most biologically relevant value of “K” called “Thermodynamic

64 Integration” (Verity & Nichols, 2016). Regardless of the speed gains these programs
65 offer, they are only able to use a single CPU core for their computations, which
66 means that these methods too, do not scale well with current multi-core IT
67 infrastructure.

68 Alternatively, a method to bootstrap multiple simultaneous runs of the software can
69 be used, such as the R (R Core Team, 2013) package *ParallelStructure* (Besnier &
70 Glover, 2013), or *StrAuto* (Chhatre & Emerson, 2017), which does exactly that for the
71 software STRUCTURE (Pritchard et al., 2000). *ParallelStructure*, however, has scaling
72 problems, as described in the manuscript, considerably loosing efficiency as more
73 CPU cores are used. *StrAuto* is another option that does indeed scale well with the
74 number of CPU cores used, but like *ParallelStructure*, it only works as a wrapper for
75 the software STRUCTURE, and cannot be used to speed up other popular genetic
76 clustering programs.

77 Furthermore, after the clustering step is finished, it is necessary to infer the number
78 of clusters that make most biological sense for the data (Earl & vonHoldt, 2012),
79 using methods such as the “Evanno test” (Evanno, Regnaut, & Goudet, 2005), or the
80 “Thermodynamic Integration” (TI) method (Verity & Nichols, 2016). After this, it is also
81 often necessary to plot the “meanQ” values of each cluster per individual, to be able
82 to interpret the biological significance of the data. This is usually done with software
83 such as DISTRUCT (ROSENBERG, 2004).

84 All of these steps typically require parsing the results files of each clustering run and
85 manually running all the required steps until the final outcome is produced (Earl &
86 vonHoldt, 2012). This is not only time consuming as it is also error prone due to the

large number of separate steps that must be taken during the process. Neither *ParallelStructure* nor *StrAuto* provide an automated and reproducible way to perform this task.

Part of this process is largely facilitated by the program STRUCTURE HARVESTER (Earl & vonHoldt, 2012), which automates the parsing of STRUCTURE runs and uses that information to perform an “Evanno test” on the data, which uses some heuristics to predict which value of ‘K’ makes the most biological sense regarding the analysed data. Although this is a very convenient automation, it still relies on manual user intervention to input the data from STRUCTURE, does not provide assistance with the plotting of the “meanQ” values and only works for the software STRUCTURE. Other programs, such as FASTSTRUCTURE include the necessary software to perform these tests, and even to plot the “meanQ” values, but still require manual intervention between these steps. *MaverickK* goes further and presents the full posterior distribution for ‘K’ using the “Thermodynamic Integration” test as an automatic last step of the analysis and even recommends some scripts for drawing “meanQ” plots, but this last step also requires human intervention.

To address these two problems (reducing run times and automating the analyses tasks), we herein present *Structure_threader*: a program to parallelize STRUCTURE, FASTSTRUCTURE and *MaverickK* runs that considerably reduces the scaling problems of previous approaches and automates the entire process, - wrapping the runs, assisting in the choice of the most biologically relevant value of K, and drawing the “meanQ” plots.

109 *Structure_threader* is available on https://github.com/StuntsPT/Structure_threader.
110 For the stable (non development) versions, check the [releases](#) page, or get it from
111 [Pypi](#).

112 3. Materials & Methods

113 3.1. Program description

114 *Structure_threader*, licensed under the GPLv3, is an open source program written in
115 python (<https://www.python.org/>) that automates and parallelizes genetic clustering
116 software (STRUCTURE, FASTSTRUCTURE and *MaverickK*) runs.

117 The software was written according the “Best Practices in the Development of
118 Bioinformatics Software” (Leprevost, Barbosa, Francisco, Perez-Riverol, & Carvalho,
119 2014) and can be run on any platform where python is available, such as GNU/Linux,
120 Mac OS X and Microsoft Windows (other platforms may also work, but were not
121 tested). Additional details are available in the program's documentation.

122 All options supported by the wrapped programs can be passed to *Structure_threader*
123 as command line arguments. These are explained in detail in both the program's
124 [online documentation](#) and builtin help text.

125 Parameters are passed to the wrapped software as in their default implementations –
126 all wrapped programs take arguments from the command line, STRUCTURE also reads
127 settings from the files “mainparams” and “extraparams” and *MaverickK* from
128 “parameters.txt”.

129 After performing the parallelized runs of the wrapped software, *Structure_threader*
130 runs a slightly modified (for integrating with *Structure_threader*) version of STRUCTURE
131 HARVESTER, *chooseK.py* (Raj et al., 2014) or TI for helping identify the most
132 biologically relevant value of 'K' for any given dataset.

133 Finally, *Structure_threader* parses the results files and draws the "meanQ" plots for
134 each considered value of "K". These are drawn in both an interactive version for
135 visualization and in a static version, better suited for publication.

136 Example data files and results are provided in the program's repository.

137 3.2. Threading strategy

138 The threading strategy used in *Structure_threader* is represented in Figure 1.
139 *Structure_threader* takes the provided input file, the values of "K" to test and the
140 required number of replicates, and creates a job queue, which is sorted by
141 decreasing complexity order. After this, P child processes are spawned, (where P is
142 the number of threads made available to the software) each containing one
143 independent instance of the wrapped program. Each of these child processes takes
144 the first available job from the queue and once it is finished, its output is processed
145 by the main process for error handling and logging. The child processes are spawned
146 using python's "multiprocessing" and "subprocess" modules from the standard library.

147 3.3. Benchmarking process

148 In order to assess the gains provided by *Structure_threader*, the program was
149 benchmarked in four different systems, described in Table 1, with various

specifications. Runs were performed twice to serve as replicates (Supplementary Table 1). Run times for STRUCTURE were assessed using both *Structure_threader* v0.4.1, *ParallelStructure* v1.0 and *StrAuto* v1.0, which were then compared. FASTSTRUCTURE and *Maverick* runs were only wrapped in *Structure_threader*, since none of the other programs supports this, and compared with the default, single threaded implementation.

Usage of RAM was monitored during the benchmarking process, and it was never detected as a bottleneck on any of the systems. None of the wrapped programs is very I/O intensive (at least as far as the tested systems were concerned), meaning that the present tests were always CPU bound.

Run times were measured using *zsh*'s *time* builtin method (wall time was used), and then normalized to a "speed up" factor (Besnier & Glover, 2013) by dividing the time of the multi-core runs by the time of the single core runs, which were performed in the measured programs' default implementations.

3.4. Benchmarking STRUCTURE

3.4.1. Test dataset description

The test file used for the benchmarks consists of 100 individuals, represented by 80 SNP loci without missing data. This dataset was crafted based on data from the *1000 genomes project* (The 1000 Genomes Project Consortium, 2015) to perform the benchmarks and was constructed aiming for a size that would be neither too small,

170 which could bias the benchmarking towards very quick runs, nor too large, to avoid
171 the process taking too long to be practical.

172 This dataset was created from public data, and instructions on how to recreate it are
173 available in the documentation and in the program's repository.

174 3.4.2. Benchmark details

175 The benchmarking process consisted of running the test dataset on STRUCTURE
176 v2.3.4 for 1×10^6 MCMC iterations, and a “burnin” length of 5×10^4 , under “admixture”
177 model (for other parameters check the program’s repository). These settings were
178 performed for values of “K” varying from 1 to 4. Each value of “K” was run with 4
179 replicates, which means a total of 16 STRUCTURE runs were performed in each
180 benchmark. All these runs were performed on the default, single threaded
181 implementation and under the *Structure_threader*, *ParallelStructure* (using the
182 “parallel_structure()” implementation, which initial testing found to be faster in all
183 used machines) and *StrAuto* wrappers.

184 3.5. Benchmarking FASTSTRUCTURE

185 3.5.1. Test dataset description

186 The test dataset used for benchmarking FASTSTRUCTURE runs, is different from the
187 one used for benchmarking STRUCTURE, since this program was designed to analyse
188 larger datasets. The tested file consists of 1000 individuals, represented by 1000
189 SNP loci. Like the previous dataset, this one was also crafted from the same public

190 data from the 1000 genomes project, and instructions for recreating it are available in
191 the documentation. The used dataset itself is available in the program's repository.

192 3.5.2. Benchmark details

193 The benchmarking process consisted of running the above described dataset for
194 values of "K" from 1 to 16 for each benchmark run.

195 The average run time of both replicates was used to plot and analyse the data. Since
196 a FASTSTRUCTURE runs do not require replicates for downstream analyses, each
197 value of "K" was run only once per benchmark, which means that a total of 16
198 FASTSTRUCTURE runs were performed both in the default implementation and under
199 the *Structure_threader* wrapper.

200 3.6. Benchmarking *Maverick*

201 3.6.1. Test dataset description

202 The test file used for the *Maverick* benchmarks is the same that was used to
203 benchmark STRUCTURE, which is described above.

204 3.6.2. Benchmark details

205 The benchmarking process consisted of running the test dataset on *Maverick* v1.0.4
206 for 1×10^4 MCMC iterations, plus a "burnin" length of 2500 iterations, with 5 replicates
207 each (for other parameters check the program's repository). These settings were
208 performed for values of "K" varying from 1 to 16.

209 4. Results & Discussion

210 Using *Structure_threader* as a wrapper for all wrapped programs has yielded
211 increases in speed that scales almost linearly with the number of processes used, at
212 least as long as physical cores are concerned, as can be seen in Figure 2 and Figure
213 3.

214 Considering the benchmark results present in Figure 2, it is clear that both
215 *Structure_threader* and *StrAuto* are more efficient methods to perform multiple
216 STRUCTURE runs on multiple core systems than *ParallelStructure* (on average 7%
217 faster in the tested systems, varying from 1% to 25% faster). *Structure_threader* is
218 also always faster than *StrAuto*, but by much smaller margins than when compared
219 with *ParallelStructure* (on average 3% faster, varying from 0.3% to 7% faster).
220 Regardless of the tested system and number of cores used, the differences in “speed
221 up” are always in favour of *Structure_threader*. When compared to *ParallelStructure*,
222 the difference increases with the requested scaling – the more physical cores are
223 used, the better the relative performance of *Structure_threader*. Also worth noting is
224 that the “speed up” values obtained here with *ParallelStructure* when using physical
225 cores, are somewhat better than what is described in (Besnier & Glover, 2013), but
226 this could be due to differences in benchmark workloads.

227 Speed up differences between *StrAuto* and *Structure_threader* are small, but can be
228 compared in Figure 2. A more detailed comparison, can be made using the data
229 tables available in Supplementary Material 1.

230 Unlike *ParallelStructure* and *StrAuto*, *Structure_threader* can also speed up the runs
231 of other programs by running them in parallel. Similar to what is done for running
232 STRUCTURE, wrapping FASTSTRUCTURE and *MaverickK* in *Structure_threader*, provides
233 considerable speed improvements, once again scaling almost linearly as long as
234 *hyper-threading* is not in effect (Figure 3).

235 Although ideally the “speed up” factor should scale linearly with the number of used
236 physical cores, this does not always happen in practice (Figure 2 and Figure 3). Of
237 the three tested wrappers, *Structure_threader* scales the closest to linearly, even
238 when using 8 physical cores, where the “speed up” factor varies between 6.24 and
239 7.91, depending on the system and the wrapped program. *ParallelStructure* shows
240 the worst scaling of the tested wrappers, especially on 8 physical threads, where the
241 “speed up” factor varies between 5.95 and 6.85.

242 The large drop in performance increase, regardless of the used wrapper program,
243 happens when using *hyper-threading* (using more than eight cores in the *Nehalem*
244 *Rack* system and more than four in the *Haswell Desktop* system – the CPUs of the
245 other two systems do not have this feature), as is sometimes described under certain
246 workloads (Leng, Ali, Hsieh, Mashayekhi, & Rooholamini, 2002; Marr et al., 2002;
247 Saini et al., 2011). We are not sure why this happens on this particular workload, but
248 the issue is not as evident when analysing smaller datasets as the one from (Besnier
249 & Glover, 2013). It is therefore hypothesised that it may be related to “cache
250 thrashing”, a phenomenon that occurs when the CPU constantly refreshes the
251 contents of L2 and L3 caches for quickly accessing different information. “Cache
252 thrashing” is more likely to happen when working with larger datasets and when

253 *hyper-threading* is active since both logical cores share L2 and L3 cache (Marr et al.,
254 2002).

255 The automated plot drawing feature of *Structure_threader* is responsible for both a
256 simplification of the analysis process (less steps per analysis), and also for the
257 reduction in random error (consequence of less human intervention).

258 The mentioned plots produced by *Structure_threader* are provided in two formats. A
259 static, vectorial image in “svg” format, especially suited for publication, and an
260 interactive HTML version of the plot, suited for results exploration.

261 5. Conclusions

262 The observed difference in efficiency between *Structure_threader* and
263 *ParallelStructure* can probably be explained by the programming languages utilized
264 in the wrappers (Python vs. R) and the fact that *ParallelStructure* solves tasks in
265 increasing order of complexity, whereas *Structure_threader* sorts them in decreasing
266 order. This strategy provides an increase in efficiency, since the sorting minimizes
267 the time each CPU core is left idle.

268 Another important difference between *ParallelStructure* and *Structure_threader* is
269 that the former is a framework to build scripts that perform the requested analyses,
270 and the latter can either be used as a framework, or directly from the command line.
271 This makes *Structure_threader* much easier to use, while simultaneously keeping the
272 same type of flexibility *ParallelStructure* offers. Although both programs can be used
273 to draw the clustering plots from the STRUCTURE results, the features offered by

274 *Structure_threader* go far beyond the basic plotting that *ParallelStructure* is capable
275 of.

276 The speed gains obtained with *Structure_threader* and *StrAuto* are very similar, with
277 only a marginal difference favouring *Structure_threader*. This difference
278 is likely due to the efficiency of *python*'s higher speed when compared to *bash*'s, and
279 eventually due to a smaller overhead of *python*'s *multiprocess* module when
280 compared to that of GNU parallel (Tange, 2011). Although both programs are run
281 from the command line interface, *Structure_threader* is more user friendly than
282 *StrAuto*, since it includes built-in help, handles user errors, and allows for a lot of
283 parameters to be defined directly in the command line.

284 *Structure_threader* was designed to exploit the power of multi-core machines for
285 speeding up multiple genetic clustering software runs, with emphasis on scalability.
286 Our results demonstrate that in every tested scenario this goal is fulfilled in a more
287 efficient way than previous solutions.

288 Furthermore, *Structure_threader* goes much farther than the two previous solutions
289 in it's capabilities to perform tests for estimating the most biologically relevant "K"
290 value, as well as plotting flexibility.

291 Although the automation process that *Structure_threader* provides does not decrease
292 computation time, it should significantly speed up the analyses process, due to the
293 human time that is saved. Furthermore, this automation is also one important step for
294 reproducibility of the studies that use this software. That being said, it is also
295 important that users interact with and explore the options and parametrization the

296 wrapped programs offer. It is critical that these are well understood in order to obtain
297 meaningful and statistically relevant results.

298 We find that the obtained decrease in run times, allied with the ease of use and
299 automation, including that of follow up analysis, make *Structure_threader* a useful
300 tool to any investigator working with population genetics/genomics data and the best
301 current choice for performing genetic clustering analyses.

302 6. Acknowledgements

303 This study was financed by Portuguese National Funds, through FCT – Fundação
304 para a Ciência e a Tecnologia, within the projects UID/BIA/00329/2013,
305 SFRH/BD/51411/2011 and SFRH/BD/86736/2012.

306 We would further like to thank Bob Verity and an anonymous reviewer for their
307 suggestions that thoroughly improved both the software and the manuscript.

308 7. References

- Besnier, F., & Glover, K. A. (2013). ParallelStructure: A R Package to Distribute
Parallel Runs of the Population Genetics Program STRUCTURE on Multi-Core
Computers. *PLoS ONE*, 8(7). <https://doi.org/10.1371/journal.pone.0070651>
- Chhatre, V. E., & Emerson, K. J. (2017). StrAuto: automation and parallelization of
STRUCTURE analysis. *BMC Bioinformatics*, 18, 192.
<https://doi.org/10.1186/s12859-017-1593-0>
- De Barro, P. J. (2005). Genetic structure of the whitefly *Bemisia tabaci* in the Asia–
Pacific region revealed using microsatellite markers. *Molecular Ecology*,
14(12), 3695–3718. <https://doi.org/10.1111/j.1365-294X.2005.02700.x>

- Earl, D. A., & vonHoldt, B. M. (2012). STRUCTURE HARVESTER: a website and program for visualizing STRUCTURE output and implementing the Evanno method. *Conservation Genetics Resources*, 4(2), 359–361.
<https://doi.org/10.1007/s12686-011-9548-7>
- Evanno, G., Regnaut, S., & Goudet, J. (2005). Detecting the number of clusters of individuals using the software structure: a simulation study. *Molecular Ecology*, 14(8), 2611–2620. <https://doi.org/10.1111/j.1365-294X.2005.02553.x>
- Lamaze, F. C., Sauvage, C., Marie, A., Garant, D., & Bernatchez, L. (2012). Dynamics of introgressive hybridization assessed by SNP population genomics of coding genes in stocked brook charr (*Salvelinus fontinalis*). *Molecular Ecology*, 21(12), 2877–2895. <https://doi.org/10.1111/j.1365-294X.2012.05579.x>
- Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., & Rooholamini, R. (2002). An Empirical Study of Hyper-Threading in High Performance Computing Clusters. In *Linux Clusters: The HPC Revolution, 2002*. St. Petersburg.
- Leprevost, F. da V., Barbosa, V. C., Francisco, E. L., Perez-Riverol, Y., & Carvalho, P. C. (2014). On best practices in the development of bioinformatics software. *Bioinformatics and Computational Biology*, 5, 199.
<https://doi.org/10.3389/fgene.2014.00199>
- Marr, D., Binns, F., Hill, D. L., Hinton, G., Koufty, D. A., Miller, A. J., & Upton, M. (2002). Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6. Retrieved from
https://www.researchgate.net/publication/237005389_Hyper-Threading_Technology_Architecture_and_Microarchitecture
- Muchadeyi, F. C., Eding, H., Wollny, C. B. A., Groeneveld, E., Makuza, S. M., Shamseldin, R., ... Weigend, S. (2007). Absence of population substructuring

- in Zimbabwe chicken ecotypes inferred using microsatellite analysis. *Animal Genetics*, 38(4), 332–339. <https://doi.org/10.1111/j.1365-2052.2007.01606.x>
- Pritchard, J. K., Stephens, M., & Donnelly, P. (2000). Inference of population structure using multilocus genotype data. *Genetics*, 155(2), 945–959.
- R Core Team. (2013). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from <http://www.R-project.org/>
- Raj, A., Stephens, M., & Pritchard, J. K. (2014). fastSTRUCTURE: Variational Inference of Population Structure in Large SNP Data Sets. *Genetics*, 197(2), 573–589. <https://doi.org/10.1534/genetics.114.164350>
- Renaut, S., Grassa, C. J., Moyers, B. T., Kane, N. C., & Rieseberg, L. H. (2012). The Population Genomics of Sunflowers and Genomic Determinants of Protein Evolution Revealed by RNAseq. *Biology*, 1(3), 575–596. <https://doi.org/10.3390/biology1030575>
- Rosenberg, N. A. (2004). distruct: a program for the graphical display of population structure. *Molecular Ecology Notes*, 4(1), 137–138. <https://doi.org/10.1046/j.1471-8286.2003.00566.x>
- Saini, S., Jin, H., Hood, R., Barker, D., Mehrotra, P., & Biswas, R. (2011). The impact of hyper-threading on processor resource utilization in production applications. In *2011 18th International Conference on High Performance Computing* (pp. 1–10). <https://doi.org/10.1109/HiPC.2011.6152743>
- Tange, O. (2011). GNU Parallel—The Command-Line Power Tool. *Login: The USENIX Magazine*, 36(1), 42–47.
- The 1000 Genomes Project Consortium. (2015). A global reference for human genetic variation. *Nature*, 526(7571), 68–74. <https://doi.org/10.1038/nature15393>

Verity, R., & Nichols, R. A. (2016). Estimating the Number of Subpopulations (K) in Structured Populations. *Genetics*, 203(4), 1827–1839.
<https://doi.org/10.1534/genetics.115.180992>

Data Accessibility

Structure_threader, the manual and example datasets are available on https://github.com/StuntsPT/Structure_threader.

Author Contributions

F. Pina-Martins has conceived the concept of the study, written most of the program code and written the manuscript. D. Silva has contributed to the ideas of the software, written code and revised the manuscript. J. Fino has contributed to the ideas of the software, written code and revised the manuscript. O. S. Paulo has mentored the work and revised the manuscript.

Supporting Information

Spreadsheet with all run times (Supplementary Table 1).

321 8.

322 **Tables**

Table 1: Characteristics of the systems where the programs were benchmarked, along with the run time of the single threaded run.

System Name	Type	CPU			OS	STRUCTURE	FASTSTRUCTURE	Maverick
		Frequency Base/Turbo (GHz)	Physical cores	Logical cores		single thread run time (s)	single thread time (s)	single thread run time (s)
Haswell Laptop	i7 4700MQ	2.4/3.4	4	8	ArchLinux	9668	3140	1009
Ivy Bridge Desktop	i5 3350P	3.1/3.3	4	4	ArchLinux	10926	2854	1140
Nehalem Rack	Xeon E5520x2	2.26/2.53	8	16	Ubuntu 16.04	16000	6019	1835
Sandy Bridge Rack	Xeon E5-2609x2	2.4	8	8	Ubuntu 12.04	15805	5054	1711

324 **Figures**

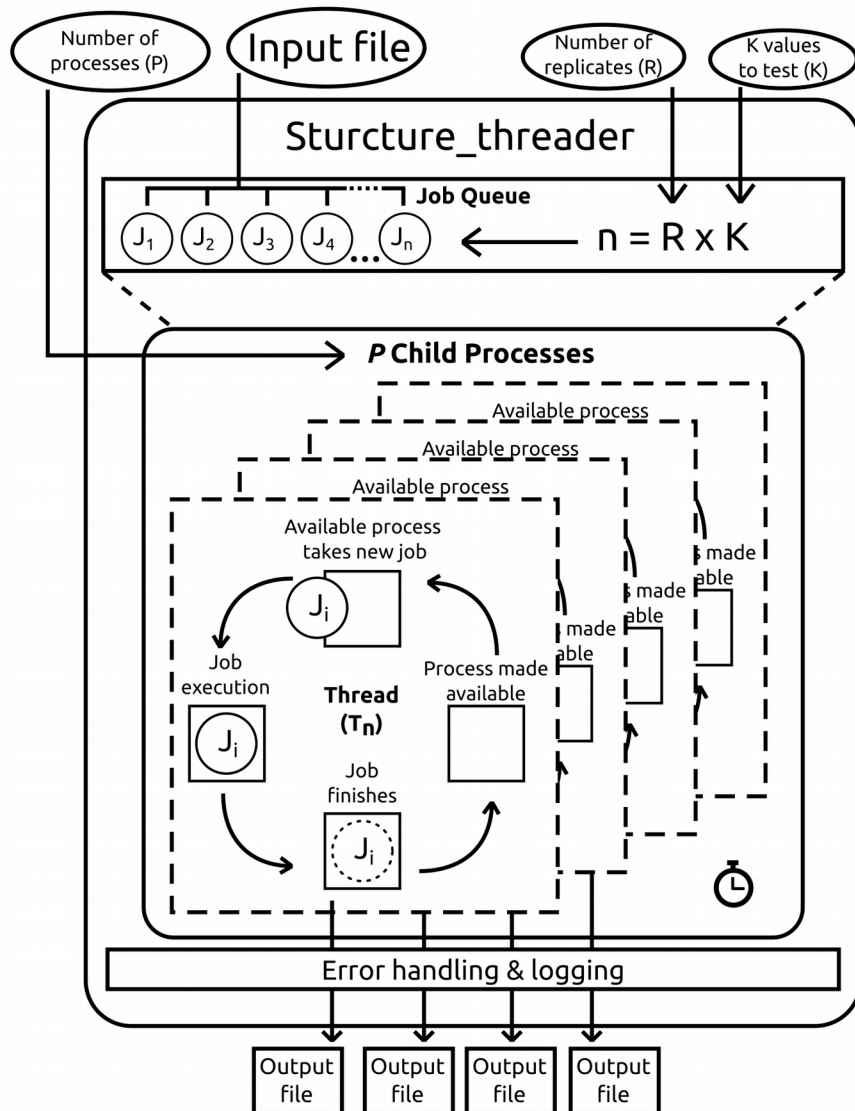


Figure 1: Threading strategy used in `Structure_threader`. Values in ellipses are read from the command line and passed to the main process, which generates a job queue. The jobs in the queue are then processed by the spawned child processes. The main process is responsible for handling and logging any errors that occur in the child processes.

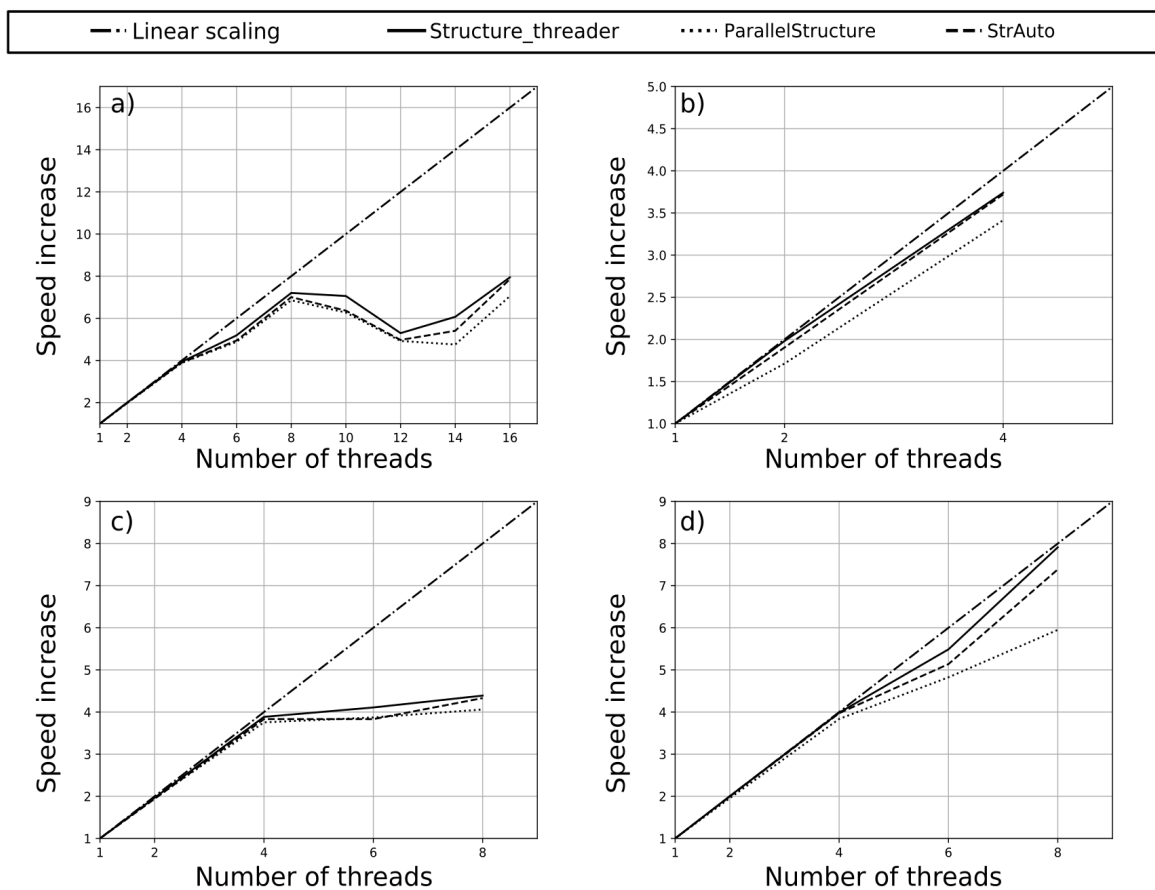


Figure 2: Plots of the *STRUCTURE* “speed up” as more threads are used in *Structure_threader*, *ParallelStructure* and *StrAuto*. Each plot represents a different system – a) is “Nehalem Rack”, b) is “Ivy Bridge Desktop”, c) is “Haswell Laptop” and d) is “Sandy Bridge Rack”.

326

327

328

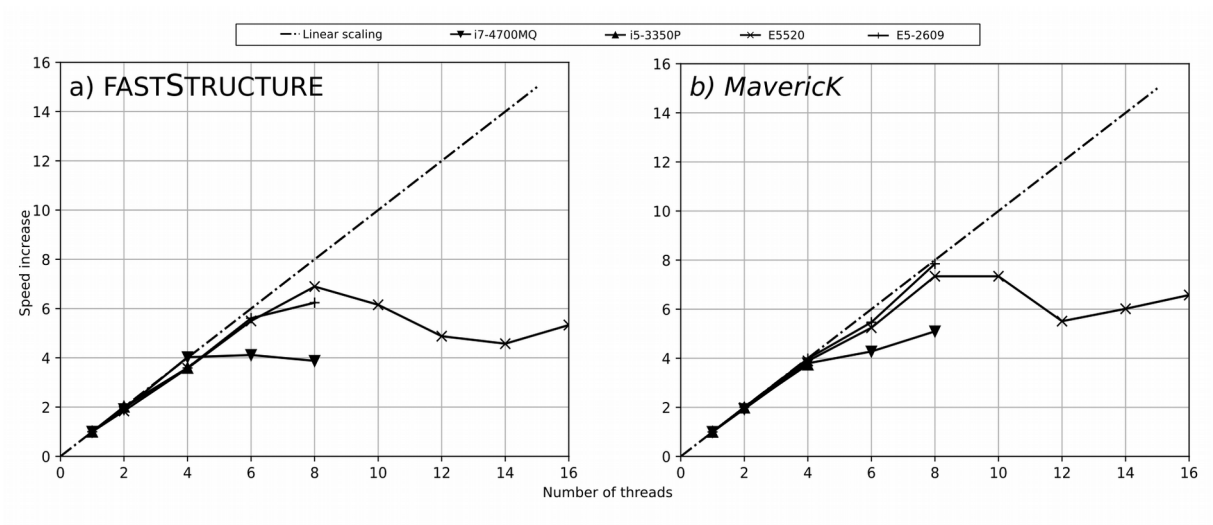


Figure 3: Plots of FASTSTRUCTURE and Maverick “speed up” as more threads are used when wrapped under Structure_threader.